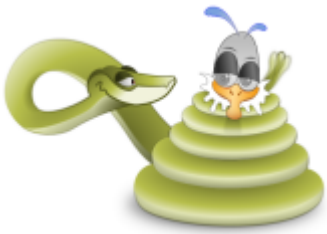
# NETWORK PROGRAMMING II - CHAT SERVER & CLIENT







## Note

Please visit /cplusplus/sockets_server_client.php for general concept for Network programming, TCP/IP/, socket, etc.

For more simpler samples, please visit basic server/client chapter: Python Network Programming

# Chat Server

In this chapter, we'll make a chat server. The server is like a middle man among clients. It can queue up to 10 clients. The server broadcasts any messages from a client to the other participants. So, the server provides a sort of chatting room.

In this chat code, the server is handling the sockets in non-blocking mode using **select.select()** method:

```
ready_to_read, ready_to_write, in_error = \
               select.select(
                  potential_readers,
                  potential_writers,
                  potential_errs,
                  timeout)
```

We pass **select()** three lists:

1. the first contains all sockets that we might want to try reading

2. the second all the sockets we might want to try writing to
3. the last (normally left empty) those that we want to check for errors

Though the **select()** itself is a blocking call (it's waiting for I/O completion), we can give it a timeout. In the code, we set **time_out = 0**, and it will poll and never block.

Actually, the **select()** function monitors all the client sockets and the server socket for readable activity. If any of the client socket is readable then it means that one of the chat client has send a message.

When the select function returns, the **ready_to_read** will be filled with an array consisting of all socket descriptors that are readable.

In the code, we're dealing with two cases:

1. If the master socket is readable, the server would accept the new connection.
2. If any of the client socket is readable, the server would read the message, and broadcast it back to all clients except the one who send the message.

# Chat Server code

Here is the server code: chat_server.py

```
# chat_server.py

import sys
import socket
import select

HOST = ''
SOCKET_LIST = []
```

```
RECV_BUFFER = 4096
PORT = 9009

def chat_server():

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind((HOST, PORT))
    server_socket.listen(10)

    # add server socket object to the list of readable connections
    SOCKET_LIST.append(server_socket)

    print "Chat server started on port " + str(PORT)

    while 1:

        # get the list sockets which are ready to be read through select
        # 4th arg, time_out  = 0 : poll and never block
        ready_to_read,ready_to_write,in_error = select.select(SOCKET_LIST,[],[]

        for sock in ready_to_read:
            # a new connection request recieved
            if sock == server_socket:
                sockfd, addr = server_socket.accept()
                SOCKET_LIST.append(sockfd)
                print "Client (%s, %s) connected" % addr

                broadcast(server_socket, sockfd, "[%s:%s] entered our chatting

            # a message from a client, not a new connection
            else:
                # process data recieved from client,
                try:
                    # receiving data from the socket.
                    data = sock.recv(RECV_BUFFER)
                    if data:
                        # there is something in the socket
                        broadcast(server_socket, sock, "\r" + '[' + str(sock.ge
                    else:
                        # remove the socket that's broken
                        if sock in SOCKET_LIST:
                            SOCKET_LIST.remove(sock)

                        # at this stage, no data means probably the connection
                        broadcast(server_socket, sock, "Client (%s, %s) is offl
```

```
                # exception
                except:
                    broadcast(server_socket, sock, "Client (%s, %s) is offline\
                    continue

    server_socket.close()

# broadcast chat messages to all connected clients
def broadcast (server_socket, sock, message):
    for socket in SOCKET_LIST:
        # send the message only to peer
        if socket != server_socket and socket != sock :
            try :
                socket.send(message)
            except :
                # broken socket connection
                socket.close()
                # broken socket, remove it
                if socket in SOCKET_LIST:
                    SOCKET_LIST.remove(socket)

if __name__ == "__main__":

    sys.exit(chat_server())
```

# On recv() & disconnection

There is an ambiguity about how we detect whether the connect is broken or not.

Here is an excerpts from http://docs.python.org/2/howto/sockets.html:

"When a **recv()** returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data successfully."

"A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That's it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes."

"But if you plan to reuse your socket for further transfers, you need to realize that there is no EOT on a socket. I repeat: if a socket send or **recv()** returns after handling 0 bytes, the connection has been broken. If the connection has not been broken, you may wait on a **recv()** forever, because the socket will not tell you that there's nothing more to read (for now)."

So, in the code, we consider the connection is off when we see no more data from the **ready_to_read** socket:

```
# process data recieved from client,
try:
    # receiving data from the socket.
    data = sock.recv(RECV_BUFFER)
    if data:
        # there is something in the socket
        broadcast(server_socket, sock, "\r" + '[' + str(sock.getpeername()) + '
    else:
        # remove the socket that's broken
        if sock in SOCKET_LIST:
            SOCKET_LIST.remove(sock)

        # at this stage, no data means probably the connection has been broken
        broadcast(server_socket, sock, "Client (%s, %s) is offline\n" % addr)
```

# Client code

Here is the client code: chat_client.py

```
# chat_client.py

import sys
import socket
import select

def chat_client():
    if(len(sys.argv) < 3) :
        print 'Usage : python chat_client.py hostname port'
        sys.exit()

    host = sys.argv[1]
    port = int(sys.argv[2])

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(2)

    # connect to remote host
    try :
        s.connect((host, port))
    except :
        print 'Unable to connect'
        sys.exit()

    print 'Connected to remote host. You can start sending messages'
    sys.stdout.write('[Me] '); sys.stdout.flush()

    while 1:
        socket_list = [sys.stdin, s]

        # Get the list sockets which are readable
        ready_to_read,ready_to_write,in_error = select.select(socket_list , [],

        for sock in ready_to_read:
            if sock == s:
                # incoming message from remote server, s
                data = sock.recv(4096)
                if not data :
                    print '\nDisconnected from chat server'
                    sys.exit()
                else :
                    #print data
                    sys.stdout.write(data)
                    sys.stdout.write('[Me] '); sys.stdout.flush()
```

```
            else :
                # user entered a message
                msg = sys.stdin.readline()
                s.send(msg)
                sys.stdout.write('[Me] '); sys.stdout.flush()

if __name__ == "__main__":

    sys.exit(chat_client())
```

The client code does either listen to incoming messages from the server or check user input. If the user types in a message then send it to the server.

We use **select()** function to handle both messages: one from **stdin** which is a user input and the other from the server. As we recall, the server side usage which is a non-blocking mode, we don't do anything on the **select()** function, and we use it as **blocking**mode. So, the **select()** function blocks (waits) till something happens. It will return only when either the server socket receives a message or the user enters a message.

# Run the code

We should run the server first:

```
$ python chat_server.py
Chat server started on port 9009
```

Then, the client code:

```
$ python chat_client.py localhost 9009
Connected to remote host. You can start sending messages
```

Note that the client should use the same port number as the server does.

Here are the output from a sample run:

```
// server terminal
$ python chat_server.py
Chat server started on port 9009
Client (127.0.0.1, 48952) connected
Client (127.0.0.1, 48953) connected
Client (127.0.0.1, 48954) connected


// client 1 terminal
$ python chat_client.py localhost 9009
Connected to remote host. You can start sending messages
[Me] [127.0.0.1:48953] entered our chatting room
[Me] [127.0.0.1:48954] entered our chatting room
[Me] client 1
[('127.0.0.1', 48953)] client 2
[('127.0.0.1', 48954)] client 3
[Me] Client (127.0.0.1, 48954) is offline
[Me]


// client 2 terminal
$ python chat_client.py localhost 9009
Connected to remote host. You can start sending messages
[Me] [127.0.0.1:48953] entered our chatting room
[Me] [127.0.0.1:48954] entered our chatting room
[Me] client 1
[('127.0.0.1', 48953)] client 2
[('127.0.0.1', 48954)] client 3
[Me] Client (127.0.0.1, 48954) is offline
[Me]

// client 3 terminal
$ python chat_client.py localhost 9009
Connected to remote host. You can start sending messages
[('127.0.0.1', 48952)] client 1
[('127.0.0.1', 48953)] client 2
[Me] client 3
[Me] ^CTraceback (most recent call last):
```

```
  File "chat_client.py", line 52, in
    sys.exit(chat_client())
  File "chat_client.py", line 30, in chat_client
    read_sockets, write_sockets, error_sockets = select.select(socket_list , []
KeyboardInterrupt
```

Note that the client #3 did go off the line at the end by typing ^C

# Python Network Programming

Network Programming - Server & Client A : Basics

Network Programming - Server & Client B : File Transfer

Network Programming II - Chat Server & Client

Network Programming III - SocketServer

Network Programming IV - SocketServer Asynchronous request